Iris: An Object-Oriented Database Management System

D. H. FISHMAN, D. BEECH, H. P. CATE, E. C. CHOW, T. CONNORS, J. W. DAVIS, N. DERRETT, C. G. HOCH, W. KENT, P. LYNGBAEK, B. MAHBOD, M. A. NEIMAT, T. A. RYAN, and M. C. SHAN
Hewlett-Packard Laboratories

The Iris database management system is a research prototype of a next-generation database management system (DBMS) intended to meet the needs of new and emerging database applications, including office information and knowledge-based systems, engineering test and measurement, and hardware and software design. Iris is exploring a rich set of new database capabilities required by these applications, including rich data-modeling constructs, direct database support for inference, novel and extensible data types, for example, to support graphic images, voice, text, vectors, and matrices, support for long transactions spanning minutes to many days, and multiple versions of data. These capabilities are, in addition to the usual support for permanence of data, controlled sharing, backup, and recovery.

The Iris DBMS consists of (1) a query processor that implements the Iris object-oriented data model, (2) a Relational Storage Subsystem (RSS) -like storage manager that provides access paths and concurrency control, backup, and recovery, and (3) a collection of programmatic and interactive interfaces. The data model supports high-level structural abstractions, such as classification, generalization, and aggregation, as well as behavioral abstractions. The interfaces to Iris include an object-oriented extension to SQL.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; *data types and structures*; H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.3 [**Database Management**]: Languages—*data description language (DDL)*; *data manipulation language (DML)*; *query languages*; H.2.4 [**Database Management**]: Systems—*query processing*; *transaction processing*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*relation systems*; *representation languages*; *semantic networks*; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software

General Terms: Languages

Additional Key Words and Phrases: Iris DBMS, LISP, object-oriented DBMS, OSQL persistent objects, SQL

1. INTRODUCTION

The Iris database management system is a research prototype of a next-generation database management system (DBMS). We are exploring new database features and capabilities through a series of increasingly more capable systems, of which the current Iris prototype is the first. In this paper we present a snapshot

Authors' address: Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304.

of the current sys
future implement
The Iris DBMS
applications, such
ing test and meas
require a rich set
of DBMSs. In ad
trolled sharing, ba
rich data modelin
types (graphic im
the database span
Iris DBMS is beir
Figure 1 is a de
system is the Iris
Manager impleme
category of object-
such as classificat
30, 31], as well
translates Iris que
which is then int
totally new forma
rely on the relatio
the Object Manag
The Iris Storag
subsystem. It is
System R [3]. Th
dynamic creation
"restores to savep
indexing, and buf
commands to retr
allow users to acce
a predicate over co
plans to modify an
noted above are di
Like most other
number of progra
Construction of in
that defines, inde
oriented interactiv
Object SQL, is an
SQL rather than
SQL in the datab
the extensions see
Inspector, is an e
the user to explore
the interobject con
the Inspector curre
a graphical interfa

of the current system and discuss its capabilities and those we are exploring for future implementations.

The Iris DBMS is intended to meet the needs of new and emerging database applications, such as office information and knowledge-based systems, engineering test and measurement, and hardware and software design. These applications require a rich set of capabilities that are not supported by the current generation of DBMSs. In addition to the usual requirement for permanence of data, controlled sharing, backup, and recovery, the new capabilities that are needed include rich data modeling constructs, direct database support for inference, novel data types (graphic images, voice, text, vectors, matrices), lengthy interactions with the database spanning minutes to many days, and multiple versions of data. The Iris DBMS is being designed to meet these needs.

Figure 1 is a depiction of the layered architecture of Iris. In the middle of the system is the Iris Object Manager, the query processor of the DBMS. The Object Manager implements the Iris Data Model [13, 14], which falls into the general category of object-oriented models that support high-level structural abstractions, such as classification, generalization/specialization, and aggregation [1, 7, 18, 26, 30, 31], as well as behavioral abstractions [5, 21, 28]. The query processor translates Iris queries and operations into an internal relational algebra format, which is then interpreted against the stored database. Instead of inventing a totally new formalism on which to base the correct behavior of our system, we rely on the relational algebra as our theory of computation. The capabilities of the Object Manager are discussed in Section 2.

The Iris Storage Manager is (currently) a conventional relational storage subsystem. It is very similar to the Relational Storage Subsystem (RSS) in System R [3]. The capabilities supported by the storage manager include the dynamic creation and deletion of relations, transactions with "savepoints" and "restores to savepoints," concurrency control, logging and recovery, archiving, indexing, and buffer management. It provides tuple-at-a-time processing, with commands to retrieve, update, insert, and delete tuples. Indexes and threads allow users to access the tuples of a relation in a predefined order. Additionally, a predicate over column values can be used to qualify tuples during retrieval. Our plans to modify and extend this subsystem to support the additional requirements noted above are discussed in Section 4.

Like most other database systems, Iris is designed to be accessible from any number of programming languages, and by stand-alone interactive interfaces. Construction of interfaces is made possible by a set of C language subroutines that defines, indeed *is*, the object manager interface. Currently, two lexically oriented interactive interfaces are supported. One of these, called OSQL, for Object SQL, is an object-oriented extension to SQL. We have chosen to extend SQL rather than invent a totally new language because of the prominence of SQL in the database community, and because, as we explored the possibility, the extensions seemed fairly natural. The other interactive interface, called the Inspector, is an extension of a LISP structure browser. This interface allows the user to explore interactively the Iris metadata (type) structures, as well as the interobject connection structures defined on a given Iris database. Although the Inspector currently offers only a lexical style of interface, it is a precursor to a graphical interface.

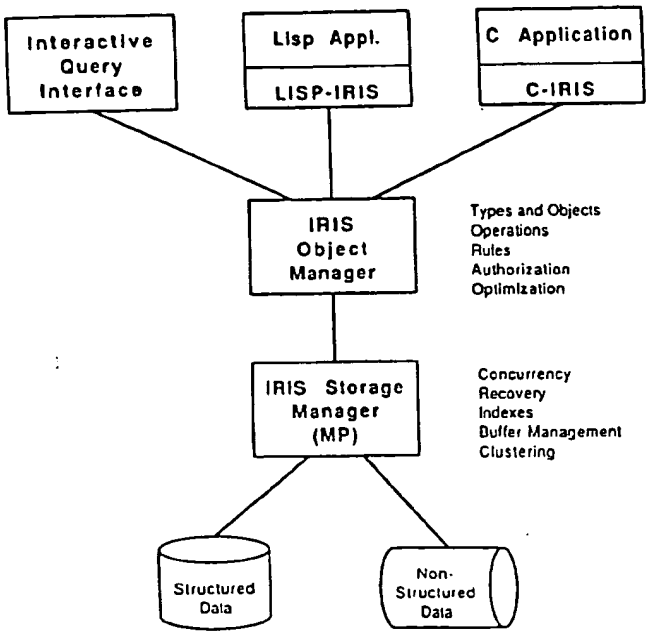


Fig. 1. Iris system structure.

As already noted, access to Iris is facilitated by a set of C language subroutines. In addition, we are exploring three kinds of LISP programmatic interfaces. The first kind is a straightforward embedding of OSQL into LISP. This has been done with minor syntactic modifications to OSQL, including a generous dose of parenthetization. Another kind of interface is the encapsulation of the Iris DBMS as a programming language *object* [10, 32, 35] whose methods correspond to the functions in the C subroutine interface to the Iris Object Manager. The third kind of interface is part of a longer term investigation into *persistent objects*, the intent of which is to make programming language objects transparently persistent and sharable across applications and languages. The various Iris interfaces are discussed in Section 3.

2. IRIS OBJECT MANAGER

The Iris Object Manager implements the Iris data model by providing support for schema definition, data manipulation, and query processing. The data model, which is based on the three constructs *objects*, *types*, and *operations*, supports inheritance and generic properties, constraints, complex or nonnormalized data, user-defined operations, version control, inference, and extensible data types. The roots of the model can be found in previous work on DAPLEX [30] and its extensions [22] and on the Taxis language [28].

2.1 Objects

Objects represent entities and concepts from the application domain being modeled. They are unique entities in the database, with their own identity and existence, and they can be referred to regardless of their attribute values.

Therefore, referen
over record-orient
can be referred to
Objects are desc
lated in terms of p
remains the same,
without affecting
abstraction and d
Objects have th

—Objects are clas
the same type.
—Objects may se
of operations.

The Iris data n
strings and numb
Literal objects ar
sented internally i
be referenced eith

or in terms of its

the

The Object Mana
nonliteral objects,
rity is supported i
if they are not bei
Note that by a "
that returns a val
attributes of Iris c
on operations.

2.2 Types and Ty

Types are named
common propertie
have a Name and
on types (see Sect
effect, therefore, t
be applicable to or
Types are orga
specialization. A t
case all instances
supertype may ha
defined on the su
properties are *inh*
The Iris type st
have multiple subt

Therefore, referential integrity [11] can be supported. This is a major advantage over record-oriented data models in which the objects, represented as records, can be referred to only in terms of their attribute values.

Objects are described by their behavior and can only be accessed and manipulated in terms of predefined operations. As long as the semantics of the operations remains the same, the database can be physically, as well as logically, reorganized without affecting application programs. This provides a very high degree of data abstraction and data independence.

Objects have the following characteristics:

—Objects are classified by type. Objects that share common properties belong to the same type.

—Objects may serve as arguments to operations and may be returned as results of operations.

The Iris data model distinguishes between *literal objects*, such as character strings and numbers, and *nonliteral* objects, such as persons and departments. Literal objects are directly representable, whereas nonliteral objects are represented internally in the database by surrogate identifiers. A nonliteral object may be referenced either in terms of its property values, for example,

the person named "Randy Newman"

or in terms of its relationships with other objects, for example,

the spouse of the person named "Sandy Newman."

The Object Manager provides operations for explicitly creating and deleting nonliteral objects, and for assigning values to their properties. Referential integrity is supported in the current prototype by allowing objects to be deleted only if they are not being referred to.

Note that by a "property" of an object we mean a *function* (a kind of operation) that returns a value when applied to that object. Thus we model properties or attributes of Iris objects with functions. This is discussed further in the section on operations.

2.2 Types and Type Hierarchies

Types are named collections of objects. Objects belonging to the same type share common properties. For example, all the objects belonging to the Person type have a Name and an Age property. Properties are operations (functions) defined on types (see Section 2.3); they are applicable to the instances of the types. In effect, therefore, types are constraints. Objects are constrained by their types to be applicable to only those properties (functions) that are defined on the types.

Types are organized in a type structure that supports generalization and specialization. A type may be declared to be the subtype of another type. In that case all instances of the subtype are also instances of the supertype. However, a supertype may have instances that do not belong to its subtype. Properties defined on the supertype are also defined on the subtype. We say that the properties are *inherited* by the subtype.

The Iris type structure is a directed acyclic graph (DAG). A given type may have multiple subtypes and multiple supertypes. Figure 2 illustrates a type graph

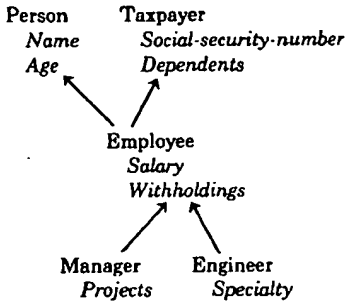


Fig. 2. A type graph.

with five types, each having a number of properties. The Employee type is a direct subtype of the Person and Taxpayer types, and the Employee type itself has two direct subtypes, Manager and Engineer.

Instances of type Employee belong to the Taxpayer and Person types as well. The properties defined on Person and on Taxpayer are inherited by Employee. Thus Employee objects have all of the six properties Salary, Withholdings, Name, Age, Social-security-number, and Dependents.

Instances of Engineer also belong to the Employee, Person, and Taxpayer types, and Engineer objects have the six properties of Employees, as well as the Specialty property. If Manager and Engineer are declared to be *disjoint*, Manager objects are guaranteed not to belong to the Engineer type. If Engineer and Manager are declared to be *overlapping*, Manager objects may belong to the Engineer type. Thus two types that are declared to be disjoint cannot be supertypes of a common subtype.

Properties may be *generic*; that is, properties defined on different types may have identical names even though their definitions may differ. Thus a database designer can introduce a property in its most general form by defining it on a general type and later refine the property definition for the more specialized subtypes. For example, the Employee type may have a general Salary property, whereas the Manager and Engineer types have Salary properties that are specific to the two job categories. This approach to design is called stepwise refinement by specialization [28]. The rules for property selection are not yet finalized.

When a generic property is applied to a given object, a single specific property must be selected at the time of application. The specific property is determined not only by the name of the generic property, but also by the type of the object to which it is applied. If the object belongs to several types that all have specific properties of the given name, the property of the most specific type is selected. If a single most specific type cannot be found, user-specified rules for property selection will apply. These rules are specified for families of functions that share the same names.

The type Object is the supertype of all other types and therefore contains every object. Types are objects themselves, and their relationships to subtypes, supertypes, and instances are expressed as functions in the system [25].

In order to support graceful database evolution, the Object Manager allows the type graph to be changed dynamically. For example, new types may be created

and existing types
lifetimes. In the c
subtypes and no
among existing ty

2.3 Operations an

An Iris operation i
are defined on typ
all Iris operations
function interchan
user-defined oper
database managen

2.3.1 *Functions*
ation consists of t
specifies the name
and results. An im
mented.

For example,

NEW FUNCTION r

declares a function
in the above exam
and the date of the

(s, d) = marriage(bol

A function may

NEW FUNCTION c

which returns the

The function d
constraints on the
For example, a fun
means that a result
be UNIQUE, whic
distinct result valu

The operation *i*
discuss below.

2.3.2 *Stored Fun*
table, mapping inp
may be implemente
The STORE oper
implemented in thi

STORE marriage

causes the Object
declaration, three c

and existing types deleted, and objects may gain or lose types throughout their lifetimes. In the current implementation a type may be deleted only if it has no subtypes and no instances. Furthermore, new subtype/supertype relationships among existing types cannot be created.

2.3 Operations and Rules

An Iris operation is a computation that may or may not return a result. Operations are defined on types and are applicable to the instances of the types. Currently all Iris operations do return results, and so we use the words *operation* and *function* interchangeably. The Iris data model and its current prototype support user-defined operations that are stored and executed under the control of the database management system.

2.3.1 *Functions for Retrieving Information.* The specification of an Iris operation consists of two parts, a *declaration* and an *implementation.* A declaration specifies the name of the operation and the number and types of its parameters and results. An implementation specifies just that, how the operation is implemented.

For example,

```
NEW FUNCTION marriage(p/Person) = (spouse/Person, date/Charstring)
```

declares a function called marriage. A function can return a compound result, as in the above example, where the result of the function contains both the spouse and the date of the marriage. This function can be called as follows:

```
(s, d) = marriage(bob)
```

A function may also return multiple results, for example,

```
NEW FUNCTION children(p/Person) = c/Person
```

which returns the set of children of a person.

The function declaration is also used to specify upper and lower bound constraints on the number of occurrences of each parameter and result value. For example, a function result value may be specified to be REQUIRED, which means that a result value must exist for each possible parameter value, or it may be UNIQUE, which means that distinct parameter values will be mapped onto distinct result values.

The operation *implementation* may be specified in various ways, which we discuss below.

2.3.2 *Stored Functions.* One way to implement a function is to store it as a table, mapping input values to their corresponding result values. Such a table may be implemented and accessed using standard relational database techniques. The STORE operation allows the user to specify that a function is to be implemented in this way. Thus

```
STORE marriage
```

causes the Object Manager to create a table with, in the case of the above declaration, three columns for the person, spouse, and date.

The mappings of several functions may be stored together in a single table. For example,

```
STORE name ON person, age ON person
```

would create a table containing persons with their names and ages. Restrictions have been introduced to ensure that such a table is in normal form.

2.3.3 *Derived Functions.* The definition of a function may be specified in terms of other functions, for example,

```
DEFINE manager(e/employee) = FIND m/employee
  WHERE m = department-manager(department(e))
```

This simple definition specifies how the manager of an employee may be derived. In general, function definitions may contain arbitrary queries. These definitions are compiled by the Object Manager into an internal relational algebra representation that is interpreted when the function is invoked.

A function definition may contain calls to several other derived functions, for example,

```
DEFINE important-manager = FIND m/employee
  WHERE FORSOME d/department
  m = department-manager(d) AND department-size(d) > 20
```

In this case the relational algebra expressions for the functions that are called in the definition are combined into a larger relational expression, with selections and joins as appropriate.

2.3.4 *Foreign Functions.* It is desirable to be able to add new data types, together with their associated operations, to a database system. For example, we might want to add a matrix or a vector data type, with associated addition and multiplication operators. In order to do this, it must be possible to link such operations into the database system and invoke them from the query processor. The current Iris prototype allows the user to define new types and operations, but only if they can be defined using the types and operations already supplied in the database system. We plan to allow privileged users to add to the database operations that are written in a traditional programming language such as C. This facility is important, both for extending the functionality of the database system and for efficient computation on rich object structures or data types.

2.3.5 *Compound Operations.* We are currently working on extensions to our model to allow the user to define operations containing sequences of operations. This requires changes to the Iris query compiler and interpreter.

2.3.6 *Rules.* Rules in the Iris model are simply functions. For example, given a *parent* function, we can define a *grandparent* function as follows:

```
DEFINE grandparent(p/Person) = FIND gp/Person
  WHERE gp = parent(parent(p))
```

A more complex rule may be defined as follows:

```
DEFINE older-cousin(p/Person) = FIND c/Person
  WHERE c = child(sibling(parent(p))) AND age(c) > age(p)
```

We note that tl
dispenses with the
from one function
bodies, if required.
An important di
Prolog as exampl
language) is that t
a stream of resul
function call retur
the inner function.

```
children(members(sa
```

returns all of the c
Like Prolog, Iri
deducible from the
The current Iris
disjunction, negati

2.3.7 *Update Op*
havior of a databas

```
SET department-ma
```

will cause the depa
invocation with the
If a function is n

```
ADD member(sales-d
```

which adds bill to t

```
REMOVE member(sa
```

The REMOVE ope
The current impl
values to be explici
specification of upd
be to increase the s
Each function in
for GET, SET, AI
implementation of
complex functions,
function definer to
can be updated.

2.3.8 *Relationship*
database operations

(1) grouping accordi
the operations o
objects and is th

We note that the nested-function notation used in Iris function definitions dispenses with the variables needed in, for example, Prolog [8], to carry results from one function call to the next. Variables can, however, be used in Iris function bodies, if required.

An important difference between C functions and Prolog rules (taking C and Prolog as examples of a traditional programming language and a rule-based language) is that the C function returns a single result, whereas the rule returns a stream of results. Iris functions can return multiple results, and a nested function call returns the concatenation of the sets of results obtained from calling the inner function. For example, the function call

children(members(sales-dept))

returns all of the children of all of the members of the sales department.

Like Prolog, Iris makes the closed-world assumption: Any fact that is not deducible from the data in the database is assumed to be false.

The current Iris prototype supports only conjunctive, nonrecursive rules, but disjunction, negation, and recursion are being studied.

2.3.7 *Update Operations.* An update operation in Iris changes the future behavior of a database function. For example, the operation

SET department-manager(sales-dept) = john

will cause the department-manager function to return the value john in a future invocation with the parameter sales-dept.

If a function is multivalued, then we can add the extra values

ADD member(sales-dept) = bill

which adds bill to the set of members of the department. Similarly, we can say

REMOVE member(sales-dept) = james

The REMOVE operation also applies to single-valued functions.

The current implementation of updates requires single argument and result values to be explicitly specified. The prototype is being extended to support the specification of updates to sets of objects. An example of such an update would be to increase the salary of all engineers by 10 percent.

Each function in Iris may have up to four compiled representations: one each for GET, SET, ADD, and REMOVE. In the case of a simple function, the implementation of the update can be deduced by the system, but for more complex functions, such as a function involving a join, it is necessary for the function definer to specify the implementation. Currently, only simple functions can be updated.

2.3.8 *Relationships and Attributes.* A database may wish to group together database operations in either of two ways:

(1) grouping according to argument types, for example, collecting together all of the operations on persons—this gives the sense of defining the properties of objects and is the traditional object-oriented approach;

(2) grouping by relationships, for example, collecting together functions and their inverses—this gives the sense of defining families of semantically related operations and is the traditional relational approach.

Either of these ways of grouping operations together is valid in its context, and the Iris data model does not insist on one or the other. Rather than introduce two concepts, however, we have chosen one.

Information about objects is modeled in Iris using *relationships*. Thus, for example, the fact that a person has a name is represented as a relationship connecting the person object and the name object. This approach is different from that of the Entity–Relationship (E–R) model [7], which allows objects to have attributes. The attribute concept is modeled in Iris by using functions whose values are derived from the relationships. Given a relationship called person–age, which connects persons and their ages, we can represent this relationship as a Boolean-valued function:

NEW FUNCTION person-age(p/Person, a/Integer) = Boolean

where person-age(John, 31) is true if the person specified has the age specified. We may then derive the functions

age(Person) = Integer
person-with-age(Integer) = Person

which are inverses of each other. The age function can be regarded as an attribute of person.

Relationships can be n-ary; for example, a relationship between mother, father, and child can be represented as a Boolean function with three parameters. An n-ary relationship can be used to derive a family of related functions, for example,

father(Person) = Person
child(Person) = Person
parents(Person) = (Person, Person)

and so on. In general, an n-ary relationship has 2^n related functions. The related functions may be derived using the DEFINE operation described previously.

2.4 Query Processing

Iris queries are implemented by compiling the queries into relational algebra representations, which are then interpreted. Some examples of queries are given in Section 3, where we discuss the interface to the Iris system.

Stored queries are implemented as functions; the query is compiled, and the compiled representation is stored in the database for later interpretation.

3. IRIS INTERFACES

The Iris DBMS may be accessed via both interactive and programmatic interfaces. These interfaces are implemented using the library of C subroutines that define the Iris Object Manager interface. The library is intended to be a platform upon which stand-alone interfaces and interfaces to various programming languages are built. In addition, programmers may use this library directly.

The following s
existing interactiv
discussed first is ir
a language extensi
extension.

3.1 Object SQL In

The initial interfac
supported by the (
For more general u
that would take th
set of property fun
to the nature of th
attributes in the E-
[9], where a row re
is also close to th
programming langu

Given the defini
simple means are
relationships such
and documents. Th
to the other usage
referring to their k
level support for re

The functional
interface for naviga
We therefore exam
IDM [4]. However
relational language
SQL to accommod
of the study, we c
feasible and fairly

The two main ex
and function model

—Direct references
may be bound to
to the objects in

—User-defined fun
SELECT clauses

We have not includ
their effect can be
understand [12].

There are also a
possible to reinterp
some of the keywor
model.

The following subsections discuss the design and functional capabilities of existing interactive and programmatic interfaces to Iris. The OSQL interface discussed first is implemented both as a stand-alone interactive interface and as a language extension. OSQL is currently embedded in Common LISP via macro extension.

3.1 Object SQL Interface

The initial interface to Iris stayed quite close to the atomic level of the operations supported by the Object Manager and was very useful in debugging the system. For more general use, however, it was decided to develop a higher level interface that would take the primitive notion of an atomic object and combine it with the set of property functions (or attributes) that the user considered to be intrinsic to the nature of the object. This is much like the treatment of entities and their attributes in the E–R model, or like one use of the tables in the relational model [9], where a row represents an object and each column represents a property. It is also close to the concept of an abstract type or class in an object-oriented programming language.

Given the definitions of two types of objects, such as Person and Document, simple means are needed to create instances of these types and to introduce relationships such as "is author of" or "has approval rights over" between persons and documents. This corresponds to the relationship sets in the E–R model and to the other usage of relational tables to relate objects (rows in other tables) by referring to their key values. Note that programming languages tend to lack high-level support for relationship sets of this kind.

The functional emphasis in Iris suggests the use of a functional style of interface for navigating around the relationships between interconnected objects. We therefore examined such languages as DAPLEX [30], GORDAS [15], and IDM [4]. However, because of the strong similarities of these languages to a relational language such as SQL [12], we also explored possible extensions to SQL to accommodate the object model and a more functional style. As a result of the study, we concluded that an Object SQL (OSQL) interface would be feasible and fairly attractive, and we decided to develop OSQL.

The two main extensions we have made beyond SQL to adapt it to the object and function model are

—Direct references to objects are used rather than their keys. Interface variables may be bound to objects on creation or retrieval and may then be used to refer to the objects in subsequent statements.

—User-defined functions and Iris system functions may appear in WHERE and SELECT clauses to give concise and powerful retrieval.

We have not included the GROUP BY and HAVING clauses on SELECT, since their effect can be achieved in other ways and they are difficult for users to understand [12].

There are also a few keyword differences from existing SQL. It should be possible to reinterpret all existing keywords mechanically, but for human users some of the keywords would be found very misleading when applied to the object model.

A few examples should illustrate both the general similarity of OSQL to SQL and the advantages of an object-based query language.

Suppose that we wish to automate some office procedures for obtaining approvals for documents. Some of the actions and corresponding OSQL statements could be as follows:

Start a new database called Approvals:

```
START Approvals;
```

Connect to the Approvals database and start a new session. This implicitly begins a new transaction:

```
CONNECT Approvals;
```

Create a new type called Person, with property functions called name, address, netaddress, and phone. Each Person object must have a value for the name function:

```
CREATE TYPE Person
  (name Charstring REQUIRED,
   address Charstring,
   netaddress Charstring,
   phone Charstring);
```

Create a new type called Approver as a subtype of Person. The type has a single property function called expertise (where we assume Topic has been created as a type), in addition to the four properties inherited from Person. The new property function is multivalued:

```
CREATE TYPE Approver SUBTYPE OF Person
  (expertise Topic MANY);
```

Create a new type called Author as a subtype of Person:

```
CREATE TYPE Author SUBTYPE OF Person;
```

Create a new type called Document:

```
CREATE TYPE Document
  (title Charstring REQUIRED,
   authorOf Author REQUIRED MANY,
   prim Topic,
   sec Topic,
   status Charstring REQUIRED,
   approverOf Approver MANY);
```

Create a stored function called grade which for a given document and a given approver returns the grade assigned to the document by the approver:

```
CREATE FUNCTION grade (Document, Approver) → Integer;
```

Create three instances of the type Approver and assign values to the property functions name (inherited from the type Person) and expertise. Bind the interface variables Smith, Jones, and Robinson to the objects created:

```
CREATE Approver (name, expertise)
INSTANCES Smith ('Albert Smith', software),
  Jones ('Isaac Jones', (finance, marketing)),
  Robinson ('Alan Robinson', (hardware, marketing, manufacturing, personnel));
```


Add the type Autl
Smith and Robinsc

```
ADD TYPE Author '
  Smith,
  Robinson;
```

Enter documents w

```
CREATE Document
INSTANCES d1 ('T
    d2 ('Workstatio
```

Assign approvers t

```
SET approverof(d1) =
```

Assign to the docu

```
SET grade(d1, Jones)
```

Make a type for ap

```
CREATE TYPE App
```

Approve the docum

```
ADD TYPE Approve
```

Commit the curren

```
COMMIT;
```

Get the title of doc

```
SELECT title(d5);
```

Get the titles of all

```
SELECT title
FOR EACH Approvec
```

Find the titles of all

```
SELECT title
FOR EACH Documen
WHERE Robinson =
```

End the current ses

```
END;
```

It is interesting tc
SQL. It would be pc
or to begin to make
move to a style tha
queries. Some of the
ward way on a relati
object manager. Mig
the path for migratic

Add the type Author to the two objects referred to by the interface variables Smith and Robinson. This shows objects being given multiple types:

```
ADD TYPE Author TO
  Smith,
  Robinson;
```

Enter documents written by Smith and Robinson:

```
CREATE Document (title, authorOf, status)
INSTANCES d1 ('The Flight from Relational', Smith, 'Received'),
    d2 ('Workstation Market Projections', Robinson, 'Received');
```

Assign approvers to the document d1:

```
SET approverof(d1) = (Jones, Robinson);
```

Assign to the document d1 the grade given by Jones:

```
SET grade(d1, Jones) = 3;
```

Make a type for approved documents:

```
CREATE TYPE ApprovedDocument SUBTYPE OF Document;
```

Approve the document d1:

```
ADD TYPE ApprovedDocument TO d1;
```

Commit the current transaction and start a new one:

```
COMMIT;
```

Get the title of document d5:

```
SELECT title(d5);
```

Get the titles of all the approved documents:

```
SELECT title
FOR EACH ApprovedDocument;
```

Find the titles of all the documents Robinson is approving:

```
SELECT title
FOR EACH Document d
WHERE Robinson = approverOf(d);
```

End the current session. This implicitly commits the current transaction:

```
END;
```

It is interesting to consider OSQL as a potential evolutionary growth path for SQL. It would be possible to use a subset of OSQL that is very similar to SQL, or to begin to make sparing use of new features such as the implicit keys, or to move to a style that takes full advantage of derived and nested functions in queries. Some of the new features of OSQL could be supported in a straightforward way on a relational system, whereas others would require a more ambitious object manager. Migration is never easy, but the OSQL approach could smooth the path for migration of both users and programs from SQL to the object world.

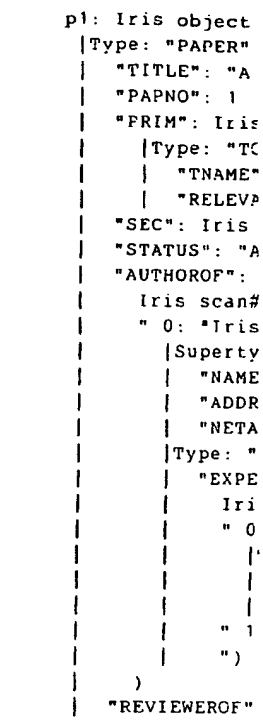
3.2 Iris Inspector

The Iris Inspector provides a mechanism for a LISP user to examine Iris database entities in the same manner in which the usual LISP values would be examined [33]. The Iris Inspector is an extension of the Inspector utility in the Hewlett-Packard Artificial Intelligence Workstation environment [2].

The Inspector is given an arbitrary LISP value and provides a *browser* (i.e., screen-oriented textual display) of the value. The user can increase or decrease the level of detail of the display. For example, at a given level of detail, the components of the value being *inspected* may be displayed only as internal names; the user can put the cursor on such a component and issue the *more detail* command, and the display of that component will be made more verbose.

The Iris Inspector provides type-specific handling of the Iris types in much the same manner as the basic Inspector provides special handling for the primitive types from which LISP Objects are built.

An example of an Inspector display is shown in Figure 3.



3.3 Iris Database Object

An object-oriented interface to Iris from Common LISP that presents the model of an *Iris database object* to the LISP programmer has been implemented. The various entities in the interface are implemented as LISP Objects [32], the methods of which are the C functions in the subroutine library defining the Object Manager interface. These types encapsulate various state information that is needed to support the methods but that does not need to be exposed to the user, such as the underlying byte patterns of the database data structures. For example, there is a *:find* method on *iris-db* that returns an object of type *iris-scan*, which, in turn, has a *:next* method to fetch the items referred to by the scan, and so forth. The arguments to all of these methods are normal LISP values; for example, the predicate for a *:find* is a list that looks like a LISP form, with the semantic difference being that the functions are Iris, not LISP, functions.

The Iris Database Object interface presents to the LISP user a family of types and their methods, which can be manipulated and examined in the same way as any other LISP Object types. This interface is in the "middle ground" of programming language/database integration, in that the database is explicitly manipulated, but the manipulation is done with the usual syntax and mechanisms of the programming language. This approach would work equally well with other object-oriented languages.

3.4 Persistent Objects

The approach of providing a database sublanguage, like our OSQL, that is embedded in host programming languages seems inappropriate if an object-oriented database is to be accessed from an object-oriented programming language. One would like to hide at least syntactic differences between in-memory and database objects, and, if possible, to hide semantic differences as well. In general, it will not be possible to hide all of the semantic differences, especially if the database is to be accessed from more than one object-oriented programming language and if the various programming languages are based on different object models. On the other hand, it is remarkable how much alike most programming

languages are whe
features. Thus it se
will be able to sup
languages, and that
between language ol
investigating this hy

Our first step in t
The interface provi
are syntactically anc
by the language. Th
that some restrictio
efficiency and simp
definition that is us

Any object-orient
for operating on obj

—the creation of a t
—the creation of a
structures;
—the creation of a r
—the application of

```
p1: Iris object "#(IRIS-HANDLE 2008,800000BA1FA814C7)"
 |Type: "PAPER"
 |  "TITLE": "A Unified Field Theory"
 |  "PAPNO": 1
 |  "PRIM": Iris object "#(IRIS-HANDLE 2008,800000A51FA814C7)"
 |    |Type: "TOPIC"
 |    |  "TNAME": "Physics"
 |    |  "RELEVANT": T
 |  "SEC": Iris object "#(IRIS-HANDLE 2008,800000A61FA814C7)"
 |  "STATUS": "A"
 |  "AUTHOROF":
 |    Iris scan#(
 |    " 0: "Iris object "#(IRIS-HANDLE 2008,800000B41FA814C7)"
 |      |Supertype of REVIEWER: "PERSON"
 |      |  "NAME": "Albert Einstein"
 |      |  "ADDRESS": NIL
 |      |  "NETADDRESS": "albert@ias.EDU"
 |      |Type: "REVIEWER"
 |      |  "EXPERTISE":
 |      |    Iris scan#(
 |      |    " 0: "Iris object "#(IRIS-HANDLE 2008,800000A51FA814C7)"
 |      |      |Type: "TOPIC"
 |      |      |  "TNAME": "Physics"
 |      |      |  "RELEVANT": T
 |      |    " 1: "Iris object "#(IRIS-HANDLE 2008,800000A61FA814C7)"
 |      |    ")
 |    )
 |  "REVIEWEROF": "(IRIS-SCAN)"
```

Figure 3

languages are when they are stripped of syntactic differences and specialist features. Thus it seems possible that a well-chosen object-oriented data model will be able to support a fairly wide variety of object-oriented programming languages, and that interfaces to these languages that hide most of the differences between language objects and database objects can be provided. We are currently investigating this hypothesis.

Our first step in this investigation was to provide a DBMS interface to LISP. The interface provides the LISP programmer with "persistent objects", which are syntactically and semantically very like the transient objects already provided by the language. There are only two differences visible to the user. The first is that some restrictions have been imposed on persistent objects for reasons of efficiency and simplicity. The other is the addition of a tag field in the type definition that is used to specify that members of the type are persistent.

Any object-oriented programming language provides four basic mechanisms for operating on objects. These are

—the creation of a type and its placement in the type hierarchy;
—the creation of an operation for one or more types, with its associated data structures;
—the creation of a new object (and its later destruction);
—the application of an operation to one or more objects.

It is therefore necessary that an object-oriented DBMS be able to support these four mechanisms in their various guises. In particular, various programming languages bundle the first and second items in various ways (e.g., Simula puts them both together, whereas they are quite separate in LISP with flavors). Thus the DBMS must provide operations for each of the individual steps that make up the creation of a type with its operations. Once these basic operations are provided by the DBMS, it is possible to put around them a syntactic layer that makes the database objects look very much like programming-language objects.

It should be noted, however, that algorithms that are appropriate for random-access memory may be highly inappropriate for disk-based storage. This means that, whatever the syntactic similarities, the programmer must be aware sometimes of whether or not an object is persistent. Furthermore, the streaming, indexing, and filtering operations that are provided by database management systems have seldom been provided by, or indeed needed by, programming languages, because their storage is truly random access and the amounts of in-memory data manipulated by programs have been relatively small. It is therefore necessary to add extra operators for persistent objects in order to provide access to these facilities. We note that some expert-systems languages, such as Prolog [8] and HPRL [29], already contain elaborate filtering and searching mechanisms (rules and inheritance), so that interfacing database searching mechanisms to such languages should be quite natural.

4. IRIS STORAGE MANAGER

The Iris prototype is built on top of a conventional relational storage manager, namely, that of Hewlett-Packard's Allbase relational DBMS. Some of the OSQL examples in Section 3.1 suggest how all instances of a type with some selected functions can be clustered in a relation. For example, all objects of type Person will be stored with their name, address, netaddress, and phone functions in one relation. The Allbase storage manager is very similar to System R's RSS [3]. Relations can be created and dropped at any time. The system supports transactions with "savepoints" and "restores to savepoints," concurrency control, logging and recovery, archiving, indexing, and buffer management. It provides tuple-at-a-time processing with commands to retrieve, update, insert, and delete tuples. Indexes and threads (links between tuples in the same relation) allow users to access the tuples of a relation in a predefined order. Additionally, a predicate over column values can be defined to qualify tuples during retrieval.

We are extending and modifying this storage subsystem to better support the Iris data model and to provide capabilities needed to support our diverse set of intended applications. Among the extensions currently being considered are support for long transactions, extensible types, and multimedia objects. We elaborate on each of these extensions in the following sections.

4.1 Transaction Management

One of the major goals of the Iris project is to provide concurrent database access to a diverse set of applications not currently well supported by existing database management systems. A characteristic of these applications is the prolonged access to and manipulation of database elements. Such interactions may last

from minutes to da
strictly two-phase
tional techniques re
concurrency would
to the transaction
such applications.

These applicatio

(1) Applications
transactions agains
likely to span seve
the arrangements f
and where the canc
of all the relevant r
transaction of long
actions against the
general effect is th
the locks on the ent
become visibile to c
rency. In such appl
cation level, for ex
compensating low-l
doing the committe

(2) AI-based app
translate into sever
discussion of the di
area and those of
applications appear
interactive nature a
active transactions
Employing a conve
cause an effectively
reduce concurrency.
the higher layers p
control mechanism
concurrency [27].

(3) Design applic
opment. Transaction
of large and complex
in the first two appl
rest being past histo
several valid states
particular design car
the requirements im
rigorous as compare

Since our initial fo
on the imposed requi

able to support these
various programming
ays (e.g., Simula puts
P with flavors). Thus
dual steps that make
basic operations are
a syntactic layer that
ing-language objects.
propriate for random-
l storage. This means
must be aware some-
more, the streaming,
atabase management
led by, programming
and the amounts of
elatively small. It is
t objects in order to
rt-systems languages,
aborate filtering and
interfacing database
atural.

nal storage manager,
S. Some of the OSQL
e with some selected
bjects of type Person
one functions in one
System R's RSS [3].
stem supports trans-
concurrency control,
agement. It provides
ate, insert, and delete
same relation) allow
rder. Additionally, a
es during retrieval.
to better support the
ort our diverse set of
being considered are
ltimedia objects. We
ons.

rrent database access
l by existing database
ons is the prolonged
nteractions may last

from minutes to days or even weeks, thereby precluding the use of conventional, strictly two-phase transaction management techniques. That is, since conventional techniques require the holding of locks until termination of the transaction, concurrency would be drastically reduced. Thus we are exploring modifications to the transaction management system that provide increased concurrency for such applications.

These applications can be categorized into three general classes:

(1) Applications in which a unit of work comprises a collection of conventional transactions against a multitude of databases, and where this unit of work is likely to span several days. A typical example of such an application would be the arrangements for a trip that might involve airline, car, and hotel reservations, and where the cancellation of the trip would require the individual cancellation of all the relevant reservations [17]. This action can be modeled as an umbrella transaction of long duration comprising several low-level conventional transactions against the airline, car rental, and hotel reservation databases. The general effect is that, at the termination of each of the low-level transactions, the locks on the entities in the respective databases are released and the changes become visibile to other independent transactions resulting in maximal concurrency. In such applications, therefore, a transaction abort or undo at the application level, for example, a trip cancellation, would result in the execution of compensating low-level transactions against the target databases, logically undoing the committed results of the previous low-level transactions.

(2) AI-based application environments whose queries against the database translate into several concurrent and interrelated transactions. An interesting discussion of the differences between the particular demands of this application area and those of the previously mentioned application and of conventional applications appears in [6]. Such an environment is characterized by its highly interactive nature and the large number of read-only applications. Such interactive transactions are potentially of moderately long duration (possibly hours). Employing a conventional transaction mechanism on a shared database will cause an effectively serialized access pattern to the database and will drastically reduce concurrency. It appears that a multilayered transaction mechanism, where the higher layers provide abstract locks and employ a different concurrency control mechanism than lower layer transactions, would provide for increased concurrency [27].

(3) Design applications such as document design, CAD, and software development. Transactions in this environment could potentially involve manipulation of large and complex objects and are likely to last several days to weeks. Although in the first two application areas there is only one valid state of the world (the rest being past history), this application area requires simultaneous existence of several valid states of the world; for example, several correct alternatives of a particular design can exist simultaneously in the database. Because of this fact, the requirements imposed on the DBMS by this application area are the most rigorous as compared with the others.

Since our initial focus is on providing support for the third area, we elaborate on the imposed requirements of these applications. Traditional databases take a

global and static view of the world. At any one time there is only one current value for any entity, and this value is changed in a very regular way. Although previous values of a particular entity may be accessible, for example, through the log file, it is the current value for the entity that is of primary importance. In contrast, design databases take a dynamic and temporal view of the world; an entity may simultaneously have several alternate values or representations in the database. Past values of an entity may be equally important to a user of a design database and may be frequently accessed.

The simultaneous presence of alternative values for a particular entity necessitates the existence of an object versioning mechanism in order to provide controlled access to these values [19, 20, 23, 24]. A version control mechanism is being explored as an integral part of the Iris Object Manager, which would form the basis for the implementation of concurrency control in this application environment. We are exploring a versioning model in which the user can create a tree of versions for any object. When we provide for merging of versions, this will be a more general version graph. The version control mechanism will dovetail with our transaction management approach, in which the user is allowed to *check out* one or more object versions for extended manipulation.

We require a new object locking mechanism that can put long-term locks on objects in persistent storage. This locking mechanism is employed at the design transaction level and is at a higher level than the traditional locks held in volatile memory by conventional transactions. This higher level lock mechanism provides a hierarchical lock structure with intention locks, as well as share and exclusive locks much like its lower level counterpart [16]. The object hierarchy and the dependencies and overlaps between object hierarchies need to be known to the lock manager so that the proper intention locks can be set.

The design database comprises a public and logically private databases. The same mechanism for long transactions also controls concurrency in the private databases [24]. When a version of an object is checked out, this version of the object, together with all the objects in its subtree, is locked in the public database and logically becomes part of a private database. The lock in the public database prevents further access to this object by others, although access to all other versions is possible. Once the object is checked out, versions of referenced objects can be made in the private database. All revisions to the private versions will be reflected in the public database at the time the entire subtree is checked back in.

On the basis of the above discussion, a multilayered transaction mechanism appears to be the appropriate solution for these diverse environments, where each application environment sees a different transaction interface. For example, a CAD application will call *checkout* to access a group of objects, whereas an AI application will call *begin_transaction* to perform a sequence of queries. We are actively evaluating the conceptual and implementational aspects of this scheme.

4.2 Extensible Types

The ability to add new data types, operations, and access methods is a desirable property of a DBMS. This ability allows one to model more easily and precisely a given application domain. For instance, a Date data type could be useful in a payroll application. In addition, this ability introduces the potential for

performance impl
can be handled di
in handling new
the movement of
frequently. Efficie
ods, for example,
type. Installing r
OEMs, and Datal

Stonebraker [3
must provide the
following:

(1) declaring the
between char
is needed at t
sentation of a
(2) defining oper
(3) implementing

Item (1) is fairly s
tasks. A syntax
presented, and the
as precedence, mu
operation must b
query interpreter
presentations of s
the parse table a
Operation installa
DBMS source cod
example, the oper
for different types
overloaded functic
an abstract data t

Item (3), allowi
linked with and i
challenge. The im
rectly with the cor
and record manag
logging and concu
tially unrelated to
of its design, we be
The interaction be
Figure 4.

The Index Man
(LM), the Buffer M
to the Log Manag
Manager. Conseq
method to unders

performance improvements. Operations on new types (e.g., subtraction of Dates) can be handled directly by the DBMS. Given the increased ability of the DBMS in handling new types, efficiency is increased, since the transfer of control and the movement of data between the application and the DBMS need not occur so frequently. Efficiency also results from the introduction of custom access methods, for example, as derived from a special collating sequence defined on a new type. Installing new types could be put to advantage by the DBMS authors, OEMs, and Database Administrators (DBAs) as well as users.

Stonebraker [34] points out that to support abstract data types, the DBMS must provide the user (most likely a DBA) with a mechanism for each of the following:

(1) declaring the existence of a new type and providing filters to translate between character strings and the new type's internal representation—this is needed at the user interface level to translate between a printable representation of a type and its internal representation;
(2) defining operations on the new types;
(3) implementing new access methods for newly created types.

Item (1) is fairly straightforward. Item (2), defining an operation, entails several tasks. A syntax for how the operation will be used in expressions must be presented, and the parser modified accordingly. Any context sensitive rules, such as precedence, must be incorporated. Additionally, a procedure to execute the operation must be presented and then stored where it can be accessed by the query interpreter. In its first implementation, we plan to require that the presentations of syntax and procedures be done at DBMS compile time so that the parse table and operation table become part of the DBMS executable. Operation installation utilities will be provided to eliminate the need to modify DBMS source code directly. Another concern is operator name overloading; for example, the operator symbol "+" may have different definitions and meanings for different types. Some of the techniques used in the Object Manager to resolve overloaded functions will be used in interpreting the meaning of an operator for an abstract data type.

Item (3), allowing the user to implement new access methods that would be linked with and interact with the existing storage manager, presents a greater challenge. The implementation of an access method interacts directly or indirectly with the concurrency control mechanism, with logging, and with the buffer and record manager. One would like to minimize the requisite interaction with logging and concurrency control, since these services are complicated and essentially unrelated to the access method from an algorithmic point of view. By virtue of its design, we believe that the Iris Storage Manager is amenable to these goals. The interaction between these modules in the Iris Storage Manager is shown in Figure 4.

The Index Manager (IM), must know how to interface to the Lock Manager (LM), the Buffer Manager (BM), and the Record Manager (RM). All interactions to the Log Manager (LG) are done through the Record Manager and the Buffer Manager. Consequently, it is not necessary for the implementor of an access method to understand the interaction with the Log Manager. In addition, the

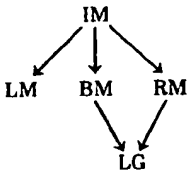


Figure 4

interface to the Lock Manager is through a single procedure that merely specifies the object requested (relation, page, or tuple) and the lock mode. We believe that this is exactly the right level of interaction with the system. The Log Manager is entirely shielded from the application programmer. Interaction with the Lock Manager is simple, yet still under access method control. This is desirable since indexing techniques may have concurrency control requirements that are less stringent than a default, system imposed method.

4.3 Multimedia Objects

In addition to such data types as Date, Money, and Matrix, office and engineering applications require the storage and manipulation of large unstructured literal types, such as text and voice data. The rigid structure of conventional DBMSs makes these systems unsuitable for multimedia applications. The Iris Object Manager plans to support vector and raster graphics, text, and voice literal types, and the Storage Manager will offer specialized storage and search solutions for processing such data types. Multimedia data will not necessarily reside on the same storage medium as the conventional data, nor will they necessarily be managed (e.g., updates logged) in the same way. Some multimedia data, for example, text, may reside in conventional files, whereas others, text or speech data, may reside on special devices, such as optical disks. Specialized hardware, for example, text search engines or voice input/output devices, may be employed to search and manipulate such data.

We envision the Iris DBMS controlling several specialized DBMSs, each dedicated to handling a specific type of data. The central DBMS knows about objects, their relationship to other objects, and associated types. A multimedia object, for example, a document, may consist of text, image, and voice subobjects. The central DBMS knows of and delegates the storage and management of these multimedia data types to the appropriate specialized DBMSs. The central DBMS also coordinates the specialized databases. A transaction spanning different types of data will begin in the central DBMS, with subtransactions spawned to each appropriate specialized DBMS. The central DBMS will coordinate the commits. The specialized DBMSs will have query processing, access methods, concurrency control, and recovery and versioning techniques that are appropriate to the data they are handling. The multimedia data types will need appropriate query interfaces and data representation and display. These will be left to application programs that interface to the central DBMS.

5. CURRENT STATUS

The Iris prototype is being implemented in C on HP-9000/320 UNIX[1] workstations. These are MC68020-based computers. The Storage Manager (still

[1] UNIX is a trademark of AT&T Bell Laboratories.

essentially unmod
Packard's Allbase
mented with parer
processor. The ex
the design stage.

The Object Mar
the model discusse
features of the m
supertypes, (atomi
side effects) have l
of other functions
only AND). Recu
mented are the fi
returned by storec
include richer oper
for these capabiliti

The interfaces tl
and Inspector inte
database object" w
Object Manager in
is the Object Man
Iris interfaces.

ACKNOWLEDGMENTS

The authors wish
suggestions for imj
take responsibility

REFERENCES

1. ABRIAL, J. R. Dat
Eds., North-Hollan
2. HP AI workstation
1985.
3. ASTRAHAN, M. M.,
GRIFFITHS, P. P., F
TRAIGER, G. R., W
system. *ACM Trans*
4. BEECH, D., AND F
Proceedings of the 9
VLDB Endowment,
5. BRODIE, M. L. On
Conference on Very
6. CAREY, M. J., DE
recovery in Prolog—
Workshop, L. Kersc
7. CHEN, P. P. The
Database Syst. 1, 1 (
8. CLOCKSIN, W. F., A
1981.
9. CODD, E. F. A rela
1970), 377-387.

essentially unmodified), called Allbase-Core, is the Storage Manager of Hewlett Packard's Allbase DBMS product. This is an RSS-like storage subsystem, augmented with parent–child links, to support both a relational and a network query processor. The extensions discussed in the Storage Manager section are still in the design stage.

The Object Manager is entirely new code. It consists of an implementation of the model discussed in Section 2 and its associated query processor. Implemented features of the model include types and type hierarchies, including multiple supertypes, (atomic) objects, and operations. Only functions (operations without side effects) have been implemented thus far. Functions may be defined in terms of other functions via function composition and Boolean combination (currently only AND). Recursive function definitions are not yet supported. Also implemented are the functors SET, ADD, and REMOVE for altering the values returned by stored functions. Capabilities that have not yet been implemented include richer operations, recursive function definitions, and versioning. Designs for these capabilities are actively being pursued.

The interfaces that have thus far been implemented for Iris include the OSQL and Inspector interactive interfaces, OSQL embedded in LISP, and the "Iris database object" whose "methods" are precisely the operations supported by the Object Manager interface. Of course, there is also the C subroutine library that *is* the Object Manager interface, the use of which is required to implement all Iris interfaces.

ACKNOWLEDGMENTS

The authors wish to thank the referees for providing numerous comments and suggestions for improving the presentation of this paper. However, the authors take responsibility for any mistakes that remain.

REFERENCES

1. ABRIAL, J. R. Data semantics. In *Data Base Management*, J. W. Klimbie, and K. L. Koffman, Eds., North-Holland, Amsterdam, 1974, pp. 1–59.
2. HP AI workstation debugger. Internal Hewlett-Packard Laboratories Rep., Palo Alto, Calif., 1985.
3. ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J. N., GRIFFITHS, P. P., KING, W. F., LORIE, R. A., MCJONES, P. R., MEHL, J. W., PUTZOLU, G. R., TRAIGER, G. R., WADE, B. W., AND WATSON, V. System R: A relational data base management system. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
4. BEECH, D., AND FELDMAN, J. S. The integrated data model—A database perspective. In *Proceedings of the 9th International Conference on Very Large Databases* (Florence, Italy, 1983). VLDB Endowment, Saratoga, Calif.
5. BRODIE, M. L. On modeling behavioral semantics of data. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, France, Sept. 9–11). ACM, New York, 1981.
6. CAREY, M. J., DEWITT, D. J., AND GRAEFE, G. Mechanisms for concurrency control and recovery in Prolog—A proposal. In *Expert Database Systems—Proceedings of the 1st International Workshop*, L. Kerschberg, Ed. The Benjamin/Cummings Publishing Co., Inc., Menlo Park, Calif.
7. CHEN, P. P. The entity-relationship model: Toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar. 1976), 9–36.
8. CLOCKSIN, W. F., AND MELLISH, C. S. *Programming in Prolog*. Springer-Verlag, New York, 1981.
9. CODD, E. F. A relational model of data for large shared data banks. *Commun. ACM 13*, 6 (June, 1970), 377–387.

10. COX, B. J. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.
11. DATE, C. J. Referential integrity. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, France, Sept. 9-11). ACM, New York, 1981.
12. DATE, C. J. *A Guide to DB2.* Addison-Wesley, Reading Mass., 1984.
13. DERRETT, N., KENT, W., AND LYNGBAEK, P. Some aspects of operations in an object-oriented database. *Database Eng. 8*, 4 (1985), 66-74.
14. DERRETT, N., FISHMAN, D. H., KENT, W., LYNGBAEK, P. AND RYAN, T. A. An object-oriented approach to data management. In *Proceedings of Compcon 31st IEEE Computer Society International Conference* (San Francisco, Calif., Mar. 1986). IEEE Computer Society Press, Washington, D.C.
15. ELMASRI, R., AND WIEDERHOLD, G. GORDAS: A formal high-level query language for the entity-relationship model. In *Entity-Relationship Approach to Information Modeling and Analysis.* P. P. Chen, Ed. Elsevier, New York, 1981.
16. GRAY, J. N. Notes on database operating systems. In *Lecture Notes in Computer Science 60, Advanced Course on Operating Systems.* R. Bayer, R. M. Graham, and G. Seegmuller, Eds. Springer-Verlag, New York, 1978.
17. GRAY, J. N. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, France, Sept. 9-11). ACM, New York, 1981.
18. HAMMER, M., AND MCLEOD, D. Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (Sept. 1981), 351-386.
19. KATZ, R. H. *Information Management for Engineering Design.* Springer-Verlag, New York, 1985.
20. KATZ, R. H., CHANG, E., AND BHATEJA, R. Version modeling concepts for computer-aided design databases. In *Proceedings of the International Conference on Management of Data* (Washington, D.C., May 28-30). ACM, New York, 1986.
21. KING, R., AND MCLEOD, D. The event database specification model. In *Proceedings of the 2d International Conference on Databases: Improving Usability and Responsiveness* (Jerusalem, Israel, June). Academic Press, New York, 1982, pp. 299-322.
22. KULKARNI, K. G. Evaluation of functional data models for database design and use. Ph.D. dissertation, Department of Computer Science, Univ. of Edinburgh, 1983.
23. LANDIS, G. S. Design evolution and history in an object-oriented CAD/CAM database. In *Proceedings of Compcon 31st IEEE Computer Society International Conference* (San Francisco, Calif., March). IEEE Computer Society Press, Washington, D.C., 1986.
24. LORIE, R., AND PLOUFFE, W. Complex objects and their use in design transactions. In *Proceedings of the Conference on Databases for Engineering Applications*, Database Week, 1983 (ACM), May 1983.
25. LYNGBAEK, P., AND KENT, W. A data modeling methodology for the design and implementation of information systems. In *Proceedings of the International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept.), IEEE Computer Society Press, Washington, D.C., 1986, pp. 6-17.
26. LYNGBAEK, P., AND MCLEOD, D. A personal data manager. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug.) VLDB Endowment, Saratoga, Calif., 1984.
27. MOSS, J. E. B., GRIFFITH, N. D., AND GRAHAM, M. H. Abstraction in recovery management. In *Proceedings of the International Conference on Management of Data* (Washington, D.C., May 28-30). ACM, New York, 1986.
28. MYLOPOULOS, J., BERNSTEIN, P. A., AND WONG, H. K. T. A language facility for designing database-intensive applications. *ACM Trans. Database Syst. 5*, 2 (June 1980), 185-207.
29. ROSENBERG, S. T. HPRL: A language for building expert systems. In *Proceedings of the International Joint Conference on Artificial Intelligence* (Karlsruhe, West Germany). William Kaufmann, Inc., Los Altos, Calif., 1983.
30. SHIPMAN, D. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (Mar. 1981), 140-173.
31. SMITH, J. M., AND SMITH, D. C. P. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst. 2*, 2 (June 1977), 105-133.
32. SNYDER, A. Com
33. STEEL, G. L. *Con*
34. STONEBRAKER, M. *2d International Co* Society Press, Was
35. STROUSTRUP, B.

Received August 1986;

32. SNYDER, A. CommonObjects: An overview. *SIGPLAN Not. 21*, 10 (Oct. 1986) 19–28.
33. STEEL, G. L. *Common Lisp: The Language.* Digital Press, Burlington, Mass., 1984.
34. STONEBRAKER, M. Inclusion of new types in relational data base systems. In *Proceedings of the 2d International Conference on Data Base Engineering* (Los Angeles, Calif., Feb.). IEEE Computer Society Press, Washington, D.C., 1986.
35. STROUSTRUP, B. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.

Received August 1986; revised September 1986; accepted November 1986